Part I. Implementation (6%):

Part 1.

```python
def bfs(start, end):
    # Begin your code (Part 1)
    '''
    At first, I read the graph by using dictionary in the form of {u :[v1, d1, s1], [v2, d2, s2], ...}
    Then, we search the graph using bfs.
    When we find a new vertex u, we push its neighbors into queue.
    Every time we pop a vertex from the queue until we find the end point; in other words, first in first out.
    This way will guarantee that we can expand our route in every direction smoothly.
    '''
    try:
        road = dict()

        with open(edgeFile) as f:
            rows = csv.reader(f)

            for row in rows:
                if row[0] == 'start':
                    continue
                road[int(row[0])] = road.get(int(row[0]), [])
                road[int(row[0])].append((int(row[1]), float(row[2]), float(row[3])))

        goback = dict()
        q = []
        q.append((start, None, 0))
        visited = set()
        visited.add(start)
        times = 0
        while(q):

            flg = False
            #cur:current, prev:previous, dist:dist
            (cur, prev, dist) = q.pop(0)
            goback[cur] = (prev, dist)
            adj = road.get(cur, [])
            for n, d, _ in adj:
                if n not in visited:
                    q.append((n, cur, d))
                    visited.add(n)
                    times += 1
                if n == end:
                    goback[n] = (cur, d)
                    flg = True
                    break
            if flg:
                break


        path = [end]
        node = end
        dist = 0

        while node != start:
            dist += goback[node][1]
            node = goback[node][0]
            path.append(node)

        path.reverse()

        return path, dist, times

    except:
        raise NotImplementedError("To be implemented")
    # End your code (Part 1)
```

Part 2.

```python
def dfs(start, end):
    # Begin your code (Part 2)
    '''
    At first, we store the graph info by dictionary again.
    Then we search end point by dfs.
    Whenever we find a new vertex, we push its neighbors into stack.
    Every time we pop the back of stack to get a new vertex; in other words, last in first out.
    This way turns out that we expand our route in only one route.
    '''
    try:
        road = dict()

        with open(edgeFile) as f:
            rows = csv.reader(f)

            for row in rows:
                if row[0] == 'start':
                    continue
                road[int(row[0])] = road.get(int(row[0]), [])
                road[int(row[0])].append((int(row[1]), float(row[2]), float(row[3])))

        goback = dict()
        stack = []
        stack.append((start, None, 0))
        visited = set()
        visited.add(start)
        times = 0

        while(stack):

            flg = False
            #cur:current, prev:previous, dist:dist
            (cur, prev, dist) = stack.pop()
            goback[cur] = (prev, dist)
            adj = road.get(cur, [])
            for n, d, _ in adj:
                if n not in visited:
                    stack.append((n, cur, d))
                    visited.add(n)
                    times += 1
                if n == end:
                    goback[n] = (cur, d)
                    flg = True
                    break
            if flg:
                break

        path = [end]
        node = end
        dist = 0

        while node != start:
            dist += goback[node][1]
            node = goback[node][0]
            path.append(node)

        path.reverse()

        return path, dist, times
    except:
        raise NotImplementedError("To be implemented")
    # End your code (Part 2)
```

Part 3.

```python
def ucs(start, end):
    # Begin your code (Part 3)
    '''
    At first, we store the graph info as usual.
    We push the start vertex into heap which can maintain the smallest distance from the start at top.
    Then, we get a new vertex from the top of heap which means the nearest vertex that doesn't be visited
    from the start vertex.
    Whenever we get a new vertex, we push its neighbors into heap with weight h+d.
    h: the distance from current to start.
    d: the distance between current and its neighbor.
    When we find the end vertex, we break the loop.
    '''
    try:
        road = dict()

        with open(edgeFile) as f:
            rows = csv.reader(f)

            for row in rows:
                if row[0] == 'start':
                    continue
                road[int(row[0])] = road.get(int(row[0]), [])
                road[int(row[0])].append((int(row[1]), float(row[2]), float(row[3])))

        goback = dict()
        heap = []
        heappush(heap, (0, start, None))
        visited, nums = set(), set()
        num, ddist = 0, 0

        while(heap):
            dist, node, baba = heappop(heap)
            if node in visited:
                continue
            visited.add(node)
            goback[node] = baba
            if node == end:
                ddist = dist
                break
            adj = road.get(node, [])
            for v, d, _ in adj:
                nums.add(v)
                if v not in visited:
                    heappush(heap, (d+dist, v, node))

        path = [end]
        node = end
        num = len(nums)

        while node != start:
            node = goback[node]
            path.append(node)

        path.reverse()

        return path, ddist, num


    except:
        raise NotImplementedError("To be implemented")
    # End your code (Part 3)
```

Part 4.

```python
def astar(start, end):
    # Begin your code (Part 4)
    '''
    This time, besides the graph info, we need to read the heuristicFile.
    From heuristicFile, we can get the distance from each vertex to the end vertex.
    Then we do the same thing as what we do in usc.
    However, this time we choose weight to be the distance from the current vertex
    to start vertex + heuristic function.
    heuristic function: the distance between current vertex and the end vertex.
    This weight can ensure us to go in the right direction and a low cost route at
    the same time.
    '''
    try:
        road, heur = dict(), dict()
        ind = 0

        with open(edgeFile) as f:
            rows = csv.reader(f)
            for row in rows:
                if row[0] == 'start':
                    continue
                road[int(row[0])] = road.get(int(row[0]), [])
                road[int(row[0])].append((int(row[1]), float(row[2]), float(row[3])))

        with open(heuristicFile) as f:
            rows = csv.reader(f)
            for row in rows:
                if row[0] == 'node':
                    for i in range(1,4):
                        if end == int(row[i]):
                            ind = i-1
                    continue
                heur[int(row[0])] = ((float(row[1]), float(row[2]), float(row[3])))

        goback = dict()
        heap = []
        heappush(heap, (heur[start][ind], start, None))
        visited, nums = set(), set()
        num, ddist = 0, 0

        while(heap):
            dist, node, baba = heappop(heap)
            dist -= heur[node][ind]
            if node in visited:
                continue
            visited.add(node)
            goback[node] = baba
            if node == end:
                ddist = dist
                break
            adj = road.get(node, [])
            for v, d, _ in adj:
                nums.add(v)
                if v not in visited:
                    heappush(heap, (d+dist+heur[v][ind], v, node))

        path = [end]
        node = end
        num = len(nums)

        while node != start:
            node = goback[node]
            path.append(node)

        path.reverse()

        return path, ddist, num

    except:
        raise NotImplementedError("To be implemented")
    # End your code (Part 4)
```

Part 6.

```python
def astar_time(start, end):
    # Begin your code (Part 6)
    '''
    To find the fastest route, we first change the weight to time consumption.
    This way, we can get the route with the least time.
    To reduce visited nodes, we use heuristic function that estimate the time
    it takes to get the end vertex.
    heuristic function =
    distance from the end vertex / the average speed from the start vertex to current one.
    '''
    try:
        road, heur = dict(), dict()
        ind = 0

        with open(edgeFile) as f:
            rows = csv.reader(f)
            for row in rows:
                if row[0] == 'start':
                    continue
                road[int(row[0])] = road.get(int(row[0]), [])
                road[int(row[0])].append((int(row[1]), float(row[2]), float(row[3])))

        with open(heuristicFile) as f:
            rows = csv.reader(f)
            for row in rows:
                if row[0] == 'node':
                    for i in range(1,4):
                        if end == int(row[i]):
                            ind = i-1
                    continue
                heur[int(row[0])] = ((float(row[1]), float(row[2]), float(row[3])))

        goback = dict()
        heap = []
        heappush(heap, (0, 0, 0, start, None))
        visited, nums = set(), set()
        num, time = 0, 0

        while(heap):
            _, sec, dist, node, baba = heappop(heap)
            if node in visited:
                continue
            visited.add(node)
            goback[node] = baba
            if node == end:
                time = sec
                break
            adj = road.get(node, [])
            for v, d, sp in adj:
                nums.add(v)
                if v not in visited:
                    t = sec + (d/sp)*3.6
                    avgs = (dist+d)/t
                    s = heur[v][ind]/avgs
                    heappush(heap, (t+s*0.8, t, dist+d, v, node))

        path = [end]
        node = end
        num = len(nums)

        while node != start:
            node = goback[node]
            path.append(node)

        path.reverse()

        return path, time, num

    except:
        raise NotImplementedError("To be implemented")
    # End your code (Part 6)
```

Part II. Results & Analysis (12%):

Results

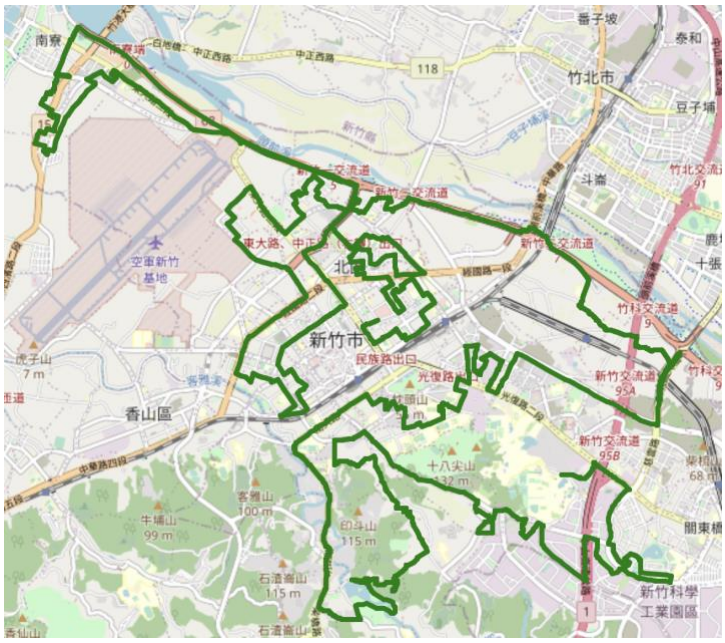
Test1: from National Yang Ming Chiao Tung University (ID: 2270143902) to Big City Shopping Mall (ID: 1079387396)

BFS:

```
The number of nodes in the path found by BFS: 88
Total distance of path found by BFS: 4978.881999999998 m
The number of visited nodes in BFS: 4273
```

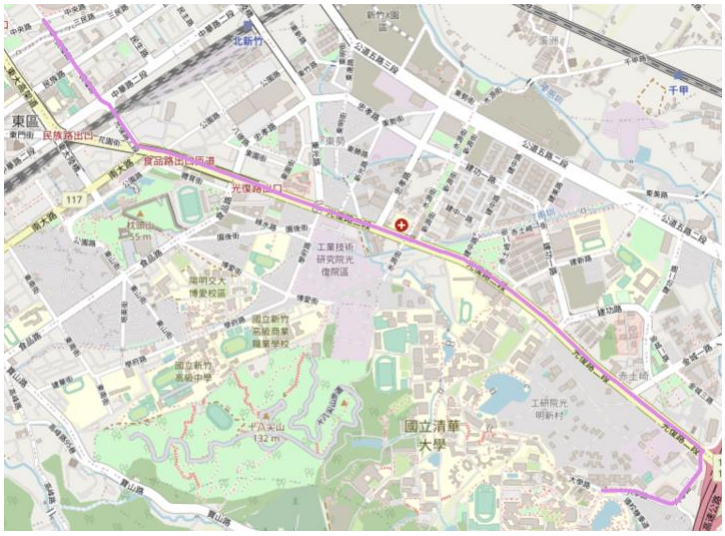
DFS (stack):

```
The number of nodes in the path found by DFS: 1718
Total distance of path found by DFS: 75504.3150000001 m
The number of visited nodes in DFS: 5235
```



UCS:

```
The number of nodes in the path found by UCS: 89
Total distance of path found by UCS: 4367.881 m
The number of visited nodes in UCS: 5158
```



A*

```
The number of nodes in the path found by A* search: 89
Total distance of path found by A* search: 4367.880999999999 m
The number of visited nodes in A* search: 312
```

A*(time)

```
The number of nodes in the path found by A* search: 93
Total second of path found by A* search: 322.0047866258889 s
The number of visited nodes in A* search: 987
```

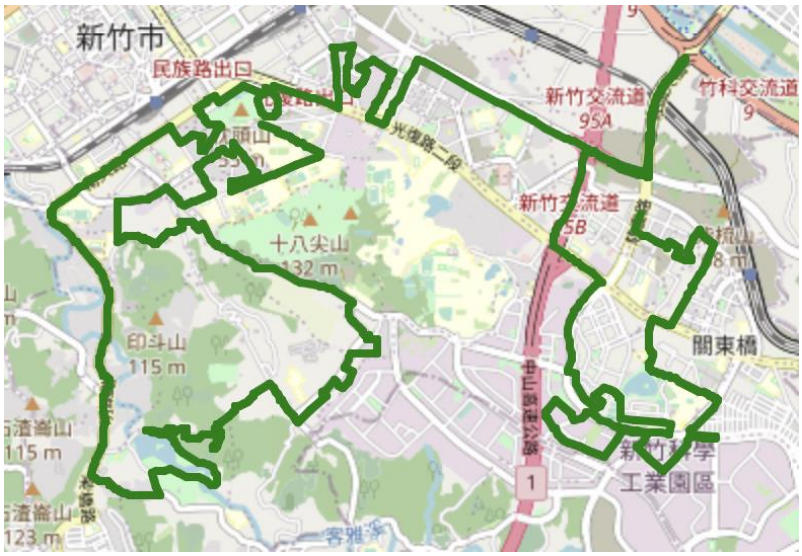
Test2: from Hsinchu Zoo (ID: 426882161) to COSTCO Hsinchu Store (ID: 1737223506)

BFS:

```
The number of nodes in the path found by BFS: 60
Total distance of path found by BFS: 4215.521000000001 m
The number of visited nodes in BFS: 4606
```

DFS(stack):

```
The number of nodes in the path found by DFS: 930
Total distance of path found by DFS: 38752.307999999895 m
The number of visited nodes in DFS: 9615
```



UCS:

```
The number of nodes in the path found by UCS: 63
Total distance of path found by UCS: 4101.84 m
The number of visited nodes in UCS: 7318
```

A*

```
The number of nodes in the path found by A* search: 63
Total distance of path found by A* search: 4101.84 m
The number of visited nodes in A* search: 1228
```

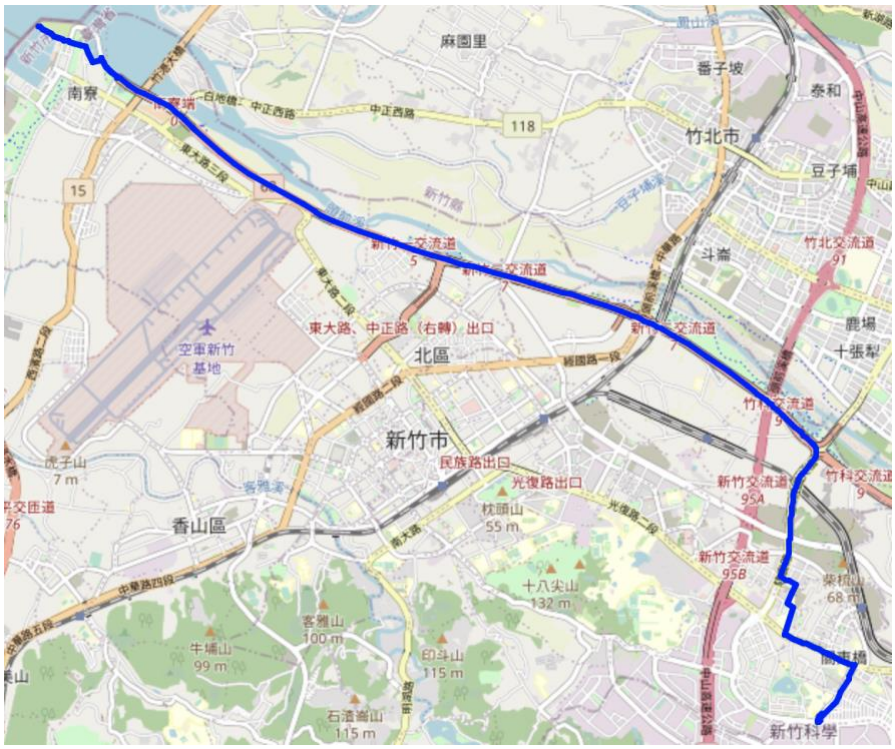
A*(time)

```
The number of nodes in the path found by A* search: 75
Total second of path found by A* search: 326.1481084327978 s
The number of visited nodes in A* search: 1969
```

Test3: from National Experimental High School At Hsinchu Science Park (ID: 1718165260) to Nanliao Fishing Port (ID: 8513026827)

BFS:

```
The number of nodes in the path found by BFS: 183
Total distance of path found by BFS: 15442.394999999995 m
The number of visited nodes in BFS: 11241
```

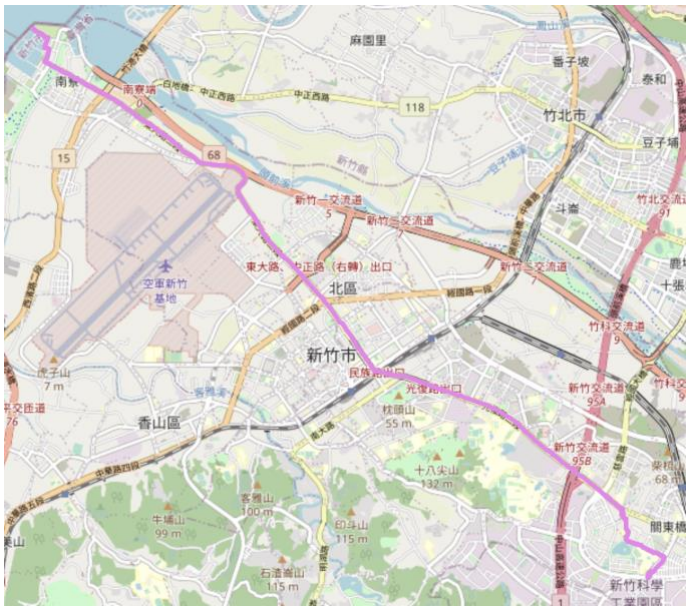


DFS(stack):

```
The number of nodes in the path found by DFS: 900
Total distance of path found by DFS: 39219.993000000024 m
The number of visited nodes in DFS: 2493
```

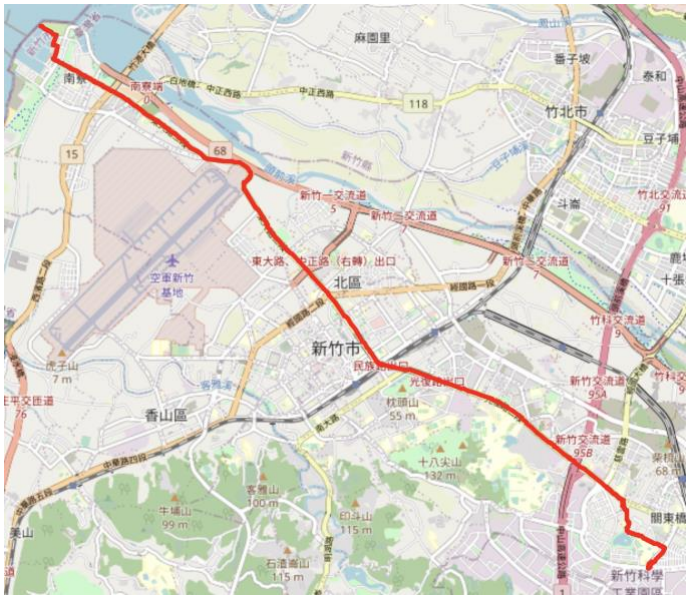
UCS:

```
The number of nodes in the path found by UCS: 288
Total distance of path found by UCS: 14212.412999999997 m
The number of visited nodes in UCS: 11934
```



A*

```
The number of nodes in the path found by A* search: 288
Total distance of path found by A* search: 14212.413000000008 m
The number of visited nodes in A* search: 7247
```



A*(time)

```
The number of nodes in the path found by A* search: 299
Total second of path found by A* search: 859.6647349934005 s
The number of visited nodes in A* search: 7026
```

Analysis

Let discuss bfs first. Because bfs uses queue, which has the first in first out property, this way will expand the possible routes in every direction. It's a little bit like a circle, and we try to enlarge its radius smoothly. This is a very safe way to find a great route to the end vertex. However, it finds it not only stably but also slowly. Second, dfs. Dfs is a quite unstable way to find the route we expect. Dfs is like a miner, it will dig into the deepest place until there is no route to dig. Then, it will go back, and dig again. Therefore, dfs quite needs luck to find a decent route we expect. In most situations, it needs to visit a great number of nodes, but find a really bad route which is long and makes no sense like our implementation results above.

Next, it's ucs. Ucs is like Dijkstra, it will always select the nearest node from start vertex and update the distances if needed. This way can ensure us to find the shortest path from start vertex to our goal by expanding the nearest node. However, it's a little bit like bfs, too. It's also stable but slow. Despite its disadvantage, it's still better than bfs for finding a great route. The last one is A star, which is best one among these searching methods at most situations. Let's see why it's so good. At first, it has the advantages of ucs because it's also using the weight containing distance while searching. But, it improves the cons of ucs by using heuristic function. Heuristic function is used to ensure us to approach our goal. Take shortest path as an example, then the heuristic function may be the distance from our goal. We add this to our weights so that we tend to choose the nodes closer to the end vertex but also has lower cost at the same time. We can find quite great performance A star searching does in finding paths above.

Last, let's talk about A star time. In this A star searching function, I replace distance with time as one of the weights. Then, I choose the heuristic function to be:

$$H(v) = \frac{d(v, end)}{d(start, v)/t(v)}$$

$$where\ d(u, v) = distance\ from\ u\ to\ v, t(v) = the\ time\ from\ start\ to\ v$$

This way, we can ensure us to find a fast route in acceptable times of visiting nodes. However, this still can't achieve our goal that we can find the route in test3 in 1000s. Therefore, I reduce its weights while searching with the cost of visiting more nodes than before.

Part III. Question Answering (12%):

1. Please describe a problem you encountered and how you solved it.

When I'm dealing with bonus part, I have difficulty reducing the result of test3 to be less than 1000s. I try many different heuristic functions, but still can't make it. In the end, I find that if I adjust the weights of heuristic function and time consumption, the result will change hugely. Therefore, I reduce the weights of heuristic function to 0.7. This can make the route I find is fast enough but also in acceptable visiting times.

2. Besides speed limit and distance, could you please come up with another attribute that is essential for route finding in the real world? Please explain the rationale.

The traffic situation is an essential condition while finding a route. If we are drivers, we will expect that we don't get stuck in a terrible traffic jam. During vacation, we can always find google map will tell you which roads are more congested than normal. Second, if the road condition is also important. Here, I'm talking about something like if this route is flat, or it may be bumpy due to land bridges or other factors. It's also important for user to decide which route to go. If I'm a bike rider, I may expect the route is almost flat so that I won't be exhausted when arriving my destination. The last, there are lots of ways to travel around or get destination. Therefore, there may be many combinations of transportation way to get there, such as walk to bus station and then take bus to SOGO is faster than riding bike there. Hence, we need to take this into consideration to provide a better route finding services.

3. As mentioned in the introduction, a navigation system involves mapping, localization, and route finding. Please suggest possible solutions for **mapping** and **localization** components?

For mapping, I think we can hire driver to drive through every route in the area we want to map it. While driving, we can detect the distance between two transactions like the format in edges.csv file. Alternatively, we can use satellites to photo the real map. Then, we can get the distance of any two nodes by calculation and construct the map by turning these photos into digital data. As for localization, we can make every position in real world has its coordinate in advance. We can get our position by sending signal to satellite. Then satellite can get our position by calculation and coordinates.

4. The estimated time of arrival (ETA) is one of the features of Uber Eats. To provide accurate estimates for users, Uber Eats needs to dynamically update based on other attributes. Please define a **dynamic heuristic** function for ETA. Please explain the rationale of your design.

To get accurate estimates, we need take many conditions into consideration. First, we can return the speed of driver periodically. If the driver gets stuck in traffic jam, then we can change our estimation according to driver's speed. Second, we can return driver's position. Although we always expect the driver to select the fastest path we find in advance. However, there are still lots of conditions that will cause the driver to select other route, such as traffic condition, a car accident or other personal reasons.